**Type Erasure**

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to achieve the following:

* Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
* Insert type casts if necessary to preserve type safety.
* Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

**Erasure of Generic Types**

During the type erasure process, the Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded or Object if the type parameter is unbounded.

Consider the following generic class that represents a node in a singly linked list:

public class Node<T> {

private T data;

private Node<T> next;

public Node(T data, Node<T> next) }

this.data = data;

this.next = next;

}

public T getData() { return data; }

// . . .

}

Because the type parameter T is unbounded, the Java compiler replaces it with Object:

public class Node {

private Object data;

private Node next;

public Node(Object data, Node next) {

this.data = data;

this.next = next;

}

public Object getData() { return data; }

// . . .

}

In the following example, the generic Node class uses a bounded type parameter:

public class Node<T extends Comparable<T>> {

private T data;

private Node<T> next;

public Node(T data, Node<T> next) {

this.data = data;

this.next = next;

}

public T getData() { return data; }

// . . .

}

The Java compiler replaces the bounded type parameter T with the first bound class, Comparable:

public class Node {

private Comparable data;

private Node next;

public Node(Comparable data, Node next) {

this.data = data;

this.next = next;

}

public Comparable getData() { return data; }

// . . .

}

**Erasure of Generic Methods**

The Java compiler also erases type parameters in generic method arguments. Consider the following generic method:

// Counts the number of occurrences of elem in anArray.

public static <T> int count(T[] anArray, T elem) {

int cnt = 0;

for (T e : anArray)

if (e.equals(elem))

++cnt

return cnt;

}

Because T is unbounded, the Java compiler replaces it with Object:

public static int count(Object[] anArray, Object elem) {

int cnt = 0;

for (Object e : anArray)

if (e.equals(elem))

++cnt;

return cnt;

}

Suppose the following classes are defined:

class Shape { /\* . . . \*/ }

class Circle extends Shape { /\* . . . \*/ }

class Rectangle extends Shape { /\* . . . \*/ }

You can write a generic method to draw different shapes:

public static <T extends Shape> void draw(T shape) { /\* . . . \*/ }

The Java compiler replaces T with Shape:

public static void draw(Shape shape) { /\* . . . \*/ }

**Effects of Type Erasure and Bridge Methods**

Sometimes type erasure causes a situation that you may not have anticipated. The following example shows how this can occur. The example shows how a compiler sometimes creates a synthetic method, called a bridge method, as part of the type erasure process.

Consider the following two classes:

public class Node<T> {

private T data;

public Node(T data) { this.data = data; }

public void setData(T data) {

System.out.println("Node.setData");

this.data = data;

}

}

public class MyNode extends Node<Integer> {

public MyNode(Integer data) { super(data); }

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

}

Now, consider the following code:

MyNode mn = new MyNode(5);

Node n = mn; // A raw type - compiler throws an unchecked warning

n.setData("Hello"); // Causes a ClassCastException to be thrown.

Integer x = mn.data;

After type erasure, this code changes as follows:

MyNode mn = new MyNode(5);

Node n = (MyNode)mn; // A raw type - compiler throws an unchecked warning

n.setData("Hello");

Integer x = (String)mn.data; // Causes a ClassCastException to be thrown.

Here is what happens as the code is executed:

* n.setData("Hello"); causes the method setData(Object) to be executed on the object of class MyNode. (The MyNode class inherited setData(Object) from Node.)
* In the body of setData(Object), the data field of the object referenced by n is assigned to aString.
* The data field of that same object, referenced via mn, can be accessed and is expected to be an integer (since mn is a MyNode, which is a Node<Integer>).
* Trying to assign a String to an Integer causes a ClassCastException from a cast inserted at the assignment by a Java compiler.

**Bridge Methods**

When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler may need to create a synthetic method, called a *bridge method*, as part of the type erasure process. You normally don’t need to worry about bridge methods, but you might be puzzled if one appears in a stack trace.

After type erasure, the Node and MyNode classes are as follows:

public class Node {

private Object data;

public Node(Object data) { this.data = data; }

public void setData(Object data) {

System.out.println("Node.setData");

this.data = data;

}

}

public class MyNode extends Node {

public MyNode(Integer data) { super(data); }

public void setData(Integer data) {

System.out.println(Integer data);

super.setData(data);

}

}

After type erasure, the method signatures do not match. The Node method becomessetData(Object) and the MyNode method becomes setData(Integer). Therefore, theMyNodesetData method does not override the NodesetData method.

To solve this problem and preserve the polymorphism of generic types after type erasure, a Java compiler generates a bridge method to ensure that subtyping works as expected. For the MyNode class, the compiler generates the following bridge method for setData:

class MyNode extends Node {

**// Bridge method generated by the compiler**

**public void setData(Object data) {**

**setData((Integer) data);**

**}**

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

// . . .

}

As you can see, the bridge method, which has the same method signature as the Node class’s setDatamethod after type erasure, delegates to the original setData method.

**Nonreifiable Types**

The “Type Erasure” section discusses the process where the compiler removes information related to type parameters and type arguments. Type erasure has consequences related to variable arguments (also known as *varargs*), methods where a vararg formal parameters contains nonreifiable type. See Chapter 4, “Passing Information to a Method or a Constructor,” for more information about varargs methods.

**Nonreifiable Types Defined**

A *reifiable type* is a type whose type information is fully available at runtime. This includes primitives, nongeneric types, raw types, and invocations of unbound wildcards.

*Nonreifiable types* are types whose information has been removed at compile time by type erasure—invocations of generic types that are not defined as unbounded wildcards. A nonreifiable type does not have all its information available at runtime. Examples of nonreifiable types are List<String> andList<Number>; the Java Virtual Machine (Java VM) cannot tell the difference between these types at runtime. As shown in the section “Restrictions on Generics” later in this chapter, there are certain situations where nonreifiable types cannot be used (e.g., in an instanceof expression or as an element in an array).

**Heap Pollution**

*Heap pollution* occurs when a variable of a parameterized type refers to an object that is not of that parameterized type. This situation occurs if the program performed some operation that gives rise to an unchecked warning at compile time. An *unchecked warning* is generated if, either at compile time (within the limits of the compile-time type checking rules) or at runtime, the correctness of an operation involving a parameterized type (e.g., a cast or method call) cannot be verified. For example, heap pollution occurs when mixing raw types and parameterized types or when performing unchecked casts.

In normal situations, when all code is compiled at the same time, the compiler issues an unchecked warning to draw your attention to potential heap pollution. If you compile sections of your code separately, it is difficult to detect the potential risk of heap pollution. If you ensure that your code compiles without warnings, then no heap pollution can occur.

**Potential Vulnerabilities of Varargs Methods with Nonreifiable Formal Parameters**

Generic methods that include vararg input parameters can cause heap pollution. Consider the followingArrayBuilder class:

public class ArrayBuilder {

public static <T> void addToList (List<T> listArg, T . . . elements)

{

for (T x : elements) {

listArg.add(x);

}

}

public static void faultyMethod(List<String> . . . l) {

Object[] objectArray = l; // Valid

objectArray[0] = Arrays.asList(42);

String s = l[0].get(0); // ClassCastException thrown here

}

}

The following example, HeapPollutionExample, uses the ArrayBuiler class:

public class HeapPollutionExample {

public static void main(String[] args) {

List<String> stringListA = new ArrayList<String>();

List<String> stringListB = new ArrayList<String>();

ArrayBuilder.addToList(stringListA, "Seven", "Eight", "Nine");

ArrayBuilder.addToList(stringListA, "Ten", "Eleven", "Twelve");

List<List<String>> listOfStringLists =

new ArrayList<List<String>>();

ArrayBuilder.addToList(listOfStringLists,

stringListA, stringListB);

ArrayBuilder.faultyMethod(Arrays.asList("Hello!"),

Arrays.asList("World!"));

}

}

When this is compiled, the following warning is produced by the definition of theArrayBuilder.addToList method:

warning: [varargs] Possible heap pollution from parameterized vararg type T

When the compiler encounters a varargs method, it translates the varargs formal parameter into an array. However, the Java programming language does not permit the creation of arrays of parameterized types. In the method ArrayBuilder.addToList, the compiler translates the varargs formal parameter T . . . elements to the formal parameter T[] elements, an array. However, because of type erasure, the compiler converts the varargs formal parameter to Object[] elements. Consequently, there is a possibility of heap pollution.

The following statement assigns the varargs formal parameter l to the Object array objectArgs:

Object[] objectArray = l;

This statement can potentially introduce heap pollution. A value that does match the parameterized type of the varargs formal parameter l can be assigned to the variable objectArray and thus can be assigned to l. However, the compiler does not generate an unchecked warning at this statement. The compiler has already generated a warning when it translated the varargs formal parameterList<String> . . . l to the formal parameter List[] l. This statement is valid; the variable l has the type List[], which is a subtype of Object[].

Consequently, the compiler does not issue a warning or error if you assign a List object of any type to any array component of the objectArray array as shown by this statement:

objectArray[0] = Arrays.asList(42);

The first array component of the objectArray array is assigned with a List object that contains one object of type Integer.

Suppose you invoke ArrayBuilder.faultyArray with the following statement:

ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));

At runtime, the Java VM throws a ClassCastException at the following statement:

// ClassCastException thrown here

String s = l[0].get(0);

The object stored in the first array component of the variable l has the type List<Integer>, but this statement is expecting an object of type List<String>.

**Prevent Warnings from Varargs Methods with Nonreifiable Formal Parameters**

If you declare a varargs method that has parameters of a parameterized type and you ensure that the body of the method does not throw a ClassCastException or other similar exception due to improper handling of the varargs formal parameter, you can prevent the warning that the compiler generates for these kinds of varargs methods by adding the following annotation to static and nonconstructor method declarations:

@SafeVarargs

The @SafeVarargs annotation is a documented part of the method’s contract; this annotation asserts that the implementation of the method will not improperly handle the varargs formal parameter.

It is also possible, though less desirable, to suppress such warnings by adding the following to the method declaration:

@SuppressWarnings({"unchecked", "varargs"})

However, this approach does not suppress warnings generated from the method’s call site. If you are unfamiliar with the @SuppressWarnings syntax, see Chapter 4, “Annotations.”